Union All Design/Implementation

I Background

1. Spark provides a union transformation which unions two RDDs and generate the result RDD.
2. In the existing Hive code, there is an existing UnionWork where a union operator is translated to a work unit. The UnionWork exists in the TezWork currently.
3. SparkWork currently consists of MapWork and ReduceWork, and the root nodes are always MapWorks.

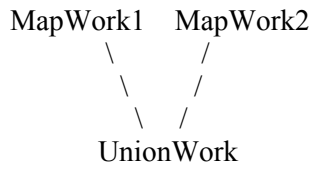
II Design Ideas

In the Hive on Spark implementation, there are a few process stages: first of all, semantic analyzer processes the hsql command and generates a operator tree; then SparkCompiler translates the operator plan into a SparkWork. SparkWork is further processed by the SparkPlanGenerator to generate a transformation graph which contains all kind of Spark transformations needed to execute the SparkWork. Finally, the transformation graph gets executed by the Spark engine and generates the result.

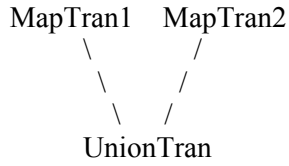
Here is a detailed example:

sample query: select name,age from voter where age < 20 union all select name, age from voter where age >40

The generated SparkWork will look like this:



From the above SparkWork, the following graph transformation is generated by SparkPlanGenerator:



Executing the above transformation graph generates the result RDD and writes the data to the temporary directory in the HDFS.

III Main classes and major APIs

UnionTran implements SparkTran

- **public** JavaPairRDD<BytesWritable, BytesWritable> transform (JavaPairRDD<BytesWritable, BytesWritable> input)
- **public void** setOtherInput(JavaPairRDD<BytesWritable, BytesWritable> otherInput)
- **public** JavaPairRDD<BytesWritable, BytesWritable> getOtherInput()

GraphTran

variables:

```
private Set<SparkTran> rootTrans // transformations on the root nodes
private Set<SparkTran> leafTrans // transformations on the leaf nodes
private Map<SparkTran, List<SparkTran>> transGraph // stores the transformation dependencies info;
private Map<SparkTran, List<SparkTran>> invertedTransGraph // stores the transformation dependencies
```

info
  private Map<SparkTran, List<JavaPairRDD<BytesWritable, BytesWritable>>> unionInputs // stores the input RDDs info for UnionTran
  private Map<SparkTran, JavaPairRDD<BytesWritable, BytesWritable>> mapInputs // stores the input RDD info for MapTran

APIs:
  public void addTran(SparkTran tran) // add a new transformation to the graph
  public void addTranInput(SparkTran tran, JavaPairRDD<BytesWritable, BytesWritable> input) // add a input RDD of a MapTran to the graph
  public void execute() // execute the graph transformation
  public void connect(SparkTran a, SparkTran b) // connect two transformation. A will be the parent of B.

SparkPlan

variables:
  private GraphTran graphTran // store graph transformation instance

APIs:
  public void execute() // execute the graphTran
  public GraphTran getTran() // returns the graphTran
  public void setTran(GraphTran tran) // set the graphTran

SparkPlanGenerator

  public SparkPlan generate(SparkWork sparkWork) // generate graph transformation from SparkWork and set the graph transformation to the SparkPlan

IV Implementation discussion

The two major implementation parts are:

1. Implementation of the *generate* API in SparkPlanGenator class. This is basically a tree/graph traversal program. Each BaseWork node is processed only once except the UnionWork. If the BaseWork is a MapWork, then generate a new MapTran and add to the GraphTran; if the BaseWork is a ReduceWork, then generate a new ReduceTran and add to the GraphTran; if the BaseWork is a UnionWork, then only generate a new UnionTran for the first time, and cache it with the UnionWork in a HashMap. The next time, when the same UnionWork is processed, fetch its UnionTran from the HashMap and create the dependency info between the UnionTran and its parent node in the graphTran. Eventually, an inverted tree structure is constructed with MapTran on the root nodes and ReduceTran/UnionTran/MapTran on other nodes.

2. Implementation of the *execute* API in GraphTran class. This is another tree/graph traversal program and transform the input RDDs according to the transformation type in each node.